

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Dokumentace projektu předmětů IFJ a IAL Implementace překladače imperativního jazyka IFJ24

Tým xgeierd00, varianta TRP-izp

Vedoucí týmu: Daniel Geier – xgeierd00 – 25%

Další členové: Patrik Mokruša – xmokrup00 – 25%

Ariana Tomen – xtomen00 – 25%

Klára Johanesová – 253003 – 25%

Obsah

1	Úvod	2
2	Práce v týmu	2
2.1	Rozdělení práce	2
3	Implementace překladače	2
3.1	Lexikální analýza	2
3.2	Syntaktická analýza	3
3.2.1	Struktura stromu	3
3.2.2	Struktura uzlů derivačního stromu	3
3.3	Sémantická analýza	4
3.4	Tabulka symbolů	4
3.5	Generování kódu	5
4	Závěr	5
5	Přílohy	6
5.1	Diagram konečného automatu lexikální analýzy	6
5.2	LL-gramatika	7
5.3	Precedenční tabulka	8
5.4	LL-tabulka	9
6	Reference	10

1 Úvod

Cílem projektu bylo vytvořit v týmu překladač imperativního jazyka IFJ24, který je podmnožinou jazyka Zig. Celý projekt je implementován v jazyce C. Dokumentace zahrnuje 4 přílohy, které obsahují Diagram konečného automatu lexikální analýzy, LL-gramatiku, Precedenční tabulku a LL-tabulku.

2 Práce v týmu

Projekt jsme vypracovali jako čtyřčlenný tým. Pro správu verzí jsme využili platformu Github. Komunikace probíhala na platformě Discord kde jsme měli časté setkání.

2.1 Rozdělení práce

- **Daniel Geier** – Generátor kódu, testování částí překladače
- **Patrik Mokruša** – Syntaktický analyzátor (rekurzivní sestup a precedenční syntaktická analýza), sémantický analyzátor, implicitní konverze ve výrazech, generátor kódu (struktura rekurzivní funkce), pomocné funkce v tabulce symbolů
- **Ariana Tomen** – Tabulka symbolů, zpracování výrazů sémantické analýzy, knihovna string
- **Klára Johanessová** – Lexikální analyzátor, dokumentace

3 Implementace překladače

Implementace překladače je rozdělena na 4 části. To jsou lexikální analýza, syntaktická analýza a sémantická analýza, a generování kódu. Každá z těchto částí je nezbytná pro správnou implementaci překladače.

Celková struktura projektu je rozdělena na 3 kroky. První je tvorba derivačního stromu, syntaktická analýza a naplnění globální tabulky symbolů. Druhý je průchod derivačním stromem při sémantické analýze. Třetí je druhý průchod derivačním stromem při generování kódu. Každá tato fáze je popsána v následujících kapitolách.

3.1 Lexikální analýza

Jako první část jsme implementovali lexikální analyzátor. Je implementován v souboru `scanner.c`, který je implementován na základě konečného stavového automatu.

Lexikální analýza funguje na principu, který načítá vstupní znaky a převede je na jednotlivé tokeny podle automatu. Tokeny jsou deklarovány v hlavičkovém souboru `scanner.h`. Jsou rozděleny do kategorií jako operátory, logické operátory, závorky, interpunkční znaky, klíčová slova, hodnoty, chyby a EOF. Každý token má vlastní strukturu a obsahuje typ a kategorii.

Hlavní funkce lexikálního analyzátoru je `scan`, je to velký `switch`, ve kterém vytváří tokeny podle jeho stavu. Bílé znaky a komentáře jsou ignorovány. Lexikální analyzátor následně pošle vytvořené tokeny syntaktickému analyzátoru pro další zpracování. Pokud funkce načte znak, který ne-

souhlasí s jazykem, tak je vrácena chybová hláška 1 a skončí. Podrobnosti o konečném stavovém automatu, jsou přiloženy v sekci přílohy.

3.2 Syntaktická analýza

Syntaktická analýza se skládá ze dvou hlavních částí, rekurzivního sestupu a precedenční analýzy.

Při analýze rekurzivním sestupem se rovnou vytváří derivační strom, který reprezentuje odvozování pravidel z LL-gramatiky. Každý uzel odpovídá terminálnímu nebo neterminálnímu uzlu a počet jeho potomků je určen pravou stranou pravidla produkce. Kořen stromu odpovídá výchozímu neterminálu (`<start>`) a listy reprezentují terminální uzly nebo epsilonové produkce.

3.2.1 Struktura stromu

- Každý uzel má 0 až 4 potomky (např. `node->first`, `node->second`, ...)
- Pokud pravidlo vede na epsilon, uzel nemá žádné potomky.
- Strom se dynamicky rozšiřuje podle pravidel gramatiky.

3.2.2 Struktura uzlů derivačního stromu

Každý uzel stromu je reprezentována strukturou `Node`:

- Typ uzlu (`NodeType`): Identifikuje, jaký neterminální nebo terminální uzel reprezentuje (např. `VariableDefine_N`, `FuncBody_N`)
- Ukazatel na potomky (`Node *first`, `Node *second`, ...)
 - Uzel může mít až 4 potomky podle pravidel gramatiky
 - Například uzel `FuncDefine_N` má čtyři potomky odpovídající pravidlu:
`<id>`, `<params_define>`, `<data_type>`, `<func_body>`
- Data (`Data_value data`): Pro specifická data uzlů (identifikátory, datové typy, hodnoty, `realace`, ...)

Precedenční analýza je implementovaná v souboru `expressionparse.c`.

Ve funkcích `Parse_expression` a `reduce`, při redukcích tvoří derivační strom od spodu nahoru.

U semantické analýzy byl problém s rozšířením `FUNEXP` (`expression` v parametru volané funkce). Zavedl jsem jenom jeden stack pro všechny rekurzivní volání, kde jednotlivé analýzy jsou oddělené akorát terminály \$.

Ošetření chyb je důležitá část syntaktického analyzátoru. Při alokaci každého nového uzlu derivačního stromu je jeho ukazatel uložen do zásobníku, který je uložen jako globální proměnná. Pokud je během analýzy nalezena syntaktická chyba, smaže se celý zásobník s uzly a následně pošle chybovou hlášku. Musel jsem zavést tento zásobník, protože vytvářím derivační strom pomocí `return` hodnot funkcí v RS.

Narazili jsme na hlavní problém, který byl konflikt v LL-tabulce. Vyřešili jsme ho zavedením dočasného ukládání tokenů, pomocí `TokenBuffer` což umožňuje jednoduchý přístup k tokenům dopředu. Používá funkci `buffercheckfirst` pro konzumování a kontrolu typu tokenu. Některé konflikty se řeší podíváním se na token dopředu a některé se řeší voláním funkce v rekurzivním sestupu, která je schopná řešit více pravidel ll-gramatiky.

3.3 Sémantická analýza

Sémantická analýza se zaměřuje na ověřování správnosti významu programu, konkrétně na kontrolu typů, kontrolu definice proměnných a platnosti výrazů. Je implementovaná v souboru `semantic.c`.

Hlavní funkcí pro provádění sémantické analýzy je `semantic_scan`, která prochází rekurzivně derivační strom, a pokud narazí na uzel, jehož typ má ve `switch case`, pak zkontroluje jeho vlastnosti podle zadání.

Pro správu identifikátorů a jejich atributů používá sémantická analýza globální symbolickou tabulku a lokální tabulku symbolů, které si rekurzivně předává v parametru funkce `semantic_scan` a postupně do ní doplňuje informace.

Pro kontrolu výrazů, které se objevují ve výrazech a operacích, používá sémantická analýza funkci `semantic_expr`, která je deklarovaná v souboru `semantic.c`. Funkce také konvertuje číselné literály jak bylo specifikované v zadání projektu. Konverze literálů probíhá od spodu nahoru a to tak, že rekurzivně projde derivační podstrom `expression`, narazí na operátor a zkontroluje, jestli se shodují. Pokud ano, nic se neděje, pokud ne, konvertuje celý podstrom na požadovaný typ.

Při ošetření chybových stavů se v globální proměnné uchovává kořen derivačního stromu. Při nalezení chyby se zavolá funkce `free_parse_tree`, která uvolní alokovanou paměť. Po uvolnění se ukončí proces s odpovídajícím chybovým kódem.

3.4 Tabulka symbolů

Pro implementaci tabulky symbolů jsme s týmem zvolili variantu TRP. Tento přístup zahrnuje použití hashovací tabulky s otevřenou adresací.

Tabulka je implementována jako dynamické pole, které používá dvojí hashovací funkcí. Tento přístup nám umožňuje efektivně spravovat symboly, i když tabulka roste, zaručuje efektivní hledání, vkládání a mazání symbolů. První hashovací funkce provádí počáteční výpočet hash hodnoty a druhá pomáhá při řešení kolizí. Tento přístup garantuje, že tabulka bude efektivní i při větších množstvích dat.

Pro zajištění správné funkce tohoto přístupu je nutné, aby velikost tabulky byla prvočíslem, což je realizované pomocí funkcí `is_prime` a `get_next_prime` (pokud nebyla počáteční hodnota prvočíslem). To výrazně zlepšuje rovnoměrné rozložení položek a snižuje počet kolizí.

Každá položka v tabulce symbolů je reprezentována strukturou **SymbolTableEntry**, která uchovává klíč (identifikátor), typ symbolu (proměnná nebo funkce) a data o proměnné nebo funkci. Tato struktura umožňuje uchovávat různé informace, jako jsou datové typy, parametry funkcí a stav symbolů (obsazen nebo smazán).

Během návrhu tabulky symbolů docházelo často ke změnám struktury kvůli rozdílům v představách o tom, jak by měla fungovat a jakou funkci by měla v našem projektu plnit. Nakonec se díky komunikaci v týmu podařilo tento problém vyřešit.

Speciální funkce implementace:

Pro každou funkci používáme spojený seznam parametrů. To nám umožňuje flexibilně přidávat parametry k funkcím bez nutnosti přerozdělování paměti při každém přidání.

Tabulka symbolů používá metodu dynamického zvětšení, kdy se její velikost zvětší dvakrát, pokud je zaplněná více než na 75%.

Pro mazání symbolů používáme logické mazání. Symboly nejsou fyzicky odstraněny okamžitě, ale označeny jako smazané, aby se předešlo poškození struktury dat při opětovném použití polí.

3.5 Generování kódu

Celý kód je implementován v souboru `generator.c` a ve hlavičkovém souboru `generator.h`. Jedná se o průběžnou generaci bez použití AST (Abstraktní syntaktický strom), kde vycházíme z vytvořeného derivačního stromu. Rekurzivně se prochází a průběžně generujeme jednotlivé konstrukce a logiky. Jednotlivé hodnoty následně bereme v daném uzlu podle LL gramatiky.

Je implementován jako jedna velká funkce `switch case`, kde ke každému typu uzlu je odpovídající `case`, kde se vygeneruje na standardní výstup. Následně obsahuje zvlášť funkce pro generování výrazů, které probíhají na zásobníku. Funkce pro převod řetězce na escape sekvenci pro `ifj` mezikód a funkce na jednotné vygenerování builtin funkcí jazyka. Výsledky funkcí se předávají přes globální proměnnou. Parametry funkce se vypisují pozpátku.

V generátoru předpokládáme že je kód validní a tudíž neřešíme některé podmínky a chybové hlášky, které řeší ostatní části překladače. Generujeme rovnou podle vygenerovaného stromu z LL gramatiky. Každý uzel podle LL gramatiky odkazuje na patřičné potomky, ze kterých bere data.

Problémy při implementaci nastaly v zamezení redefinice ve funkci `while` a řešení `funexp`. Následně byly vyřešeny a opraveny.

4 Závěr

Tento projekt se nám podařil dokončit a umožnil nám si vytvořit vlastní funkční překladač. Vyzkoušeli jsme si vytvořit projekt od návrhu až po testování. Práce na projektu nám dala mnoho zkušeností a nových vědomostí. Naučili jsme se používat různé nástroje, jako je Git pro správu verzí, a nástroje pro analýzu a testování kódu. Zároveň jsme si vylepšili schopnosti ve vzájemné spolupráci v týmu. Shodli jsme se že nám tento projekt dal velmi mnoho, nejen z hlediska technických dovedností, ale i ve smyslu týmové práce a schopnosti řešit složité problémy.

5.2 LL-gramatika

	PRAVIDLA	PREDICT
1)	$\langle \text{start} \rangle \rightarrow \langle \text{program_prolog} \rangle \langle \text{program} \rangle$	const
2)	$\langle \text{program_prolog} \rangle \rightarrow \text{const ifj} = \text{@import("ifj24.zig");}$	const
3)	$\langle \text{program} \rangle \rightarrow \langle \text{func_define} \rangle \langle \text{program} \rangle$	pub
4)	$\langle \text{program} \rangle \rightarrow \varepsilon$	EOF
5)	$\langle \text{id} \rangle \rightarrow \text{id}$	id_token
6)	$\langle \text{int} \rangle \rightarrow \text{int}$	int_token
7)	$\langle \text{float} \rangle \rightarrow \text{float}$	float_token
8)	$\langle \text{string} \rangle \rightarrow \text{string}$	str_token
9)	$\langle \text{data_type} \rangle \rightarrow \text{i32}$	i32
10)	$\langle \text{data_type} \rangle \rightarrow \text{f64}$	f64
11)	$\langle \text{data_type} \rangle \rightarrow [\text{u8}$	[
12)	$\langle \text{data_type} \rangle \rightarrow \text{void}$	void
13)	$\langle \text{func_define} \rangle \rightarrow \text{pub fn } \langle \text{id} \rangle (\langle \text{params_define} \rangle) \langle \text{data_type} \rangle \{ \langle \text{func_body} \rangle \}$	pub
14)	$\langle \text{params_define} \rangle \rightarrow \varepsilon$)
15)	$\langle \text{params_define} \rangle \rightarrow \langle \text{id} \rangle : \langle \text{data_type} \rangle \langle \text{params_define_next} \rangle$	id
16)	$\langle \text{params_define_next} \rangle \rightarrow \varepsilon$)
17)	$\langle \text{params_define_next} \rangle \rightarrow , \langle \text{id} \rangle : \langle \text{data_type} \rangle \langle \text{params_define_next} \rangle$,
18)	$\langle \text{func_body} \rangle \rightarrow \varepsilon$	}
19)	$\langle \text{func_body} \rangle \rightarrow \langle \text{statement} \rangle \langle \text{func_body} \rangle$	const, var, id, if, while, return, _
20)	$\langle \text{statement} \rangle \rightarrow \langle \text{variable_define} \rangle$	const, var
21)	$\langle \text{statement} \rangle \rightarrow \langle \text{variable_assign} \rangle$	id, _
22)	$\langle \text{statement} \rangle \rightarrow \langle \text{if} \rangle$	if
23)	$\langle \text{statement} \rangle \rightarrow \langle \text{while} \rangle$	while
24)	$\langle \text{statement} \rangle \rightarrow \langle \text{void_call} \rangle$	id
25)	$\langle \text{statement} \rangle \rightarrow \langle \text{return_statement} \rangle$	return
26)	$\text{"params_rhs"} \rightarrow \langle \text{EXPRESSION} \rangle$	int, float, id, (
27)	$\text{"params_rhs"} \rightarrow \langle \text{string} \rangle$	str
28)	$\text{"rhs"} \rightarrow \langle \text{EXPRESSION} \rangle$	int, float, id, (
29)	$\text{"rhs"} \rightarrow \langle \text{string} \rangle$	str
30)	$\langle \text{variable_define} \rangle \rightarrow \text{const } \langle \text{id} \rangle : \langle \text{data_type} \rangle = \text{"rhs"};$	const
31)	$\langle \text{variable_define} \rangle \rightarrow \text{var } \langle \text{id} \rangle : \langle \text{data_type} \rangle = \text{"rhs"};$	var
32)	$\langle \text{variable_assign} \rangle \rightarrow \langle \text{id} \rangle = \text{"rhs"};$	id
33)	$\langle \text{variable_assign} \rangle \rightarrow _ = \text{"rhs"};$	_
34)	$\langle \text{func_call} \rangle \rightarrow \langle \text{id} \rangle (\langle \text{params} \rangle)$	id
35)	$\langle \text{params} \rangle \rightarrow \varepsilon$)
36)	$\langle \text{params} \rangle \rightarrow \text{"params_rhs"} \langle \text{params_next} \rangle$	id, str, float, int, (
37)	$\langle \text{params_next} \rangle \rightarrow \varepsilon$)
38)	$\langle \text{params_next} \rangle \rightarrow , \text{"params_rhs"} \langle \text{params_next} \rangle$,
39)	$\langle \text{if} \rangle \rightarrow \text{if } (\langle \text{EXPRESSION} \rangle) \{ \langle \text{func_body} \rangle \} \text{else} \{ \langle \text{func_body} \rangle \}$	if
40)	$\langle \text{if} \rangle \rightarrow \text{if } (\langle \text{id} \rangle) \langle \text{id} \rangle \{ \langle \text{func_body} \rangle \} \text{else} \{ \langle \text{func_body} \rangle \}$	if
41)	$\langle \text{while} \rangle \rightarrow \text{while } (\langle \text{EXPRESSION} \rangle) \{ \langle \text{func_body} \rangle \}$	while
42)	$\langle \text{while} \rangle \rightarrow \text{while } (\langle \text{id} \rangle) \langle \text{id} \rangle \{ \langle \text{func_body} \rangle \}$	while
43)	$\langle \text{void_call} \rangle \rightarrow \langle \text{id} \rangle (\langle \text{params} \rangle);$	id
44)	$\langle \text{return_statement} \rangle \rightarrow \text{return};$	return
45)	$\langle \text{return_statement} \rangle \rightarrow \text{return "rhs"};$	return
46)	$\langle \text{EXPRESSION} \rangle \rightarrow \text{"zpracovano pomoci precedencni analyzy"}$	

5.3 Precedenční tabulka

	EXPR.	comp.	term	factor	>	>=	<	<=	==	!=	+	-	*	/	float	int	id	()	\$
EXPRESSION																				
compared																				
term																				
factor																				
>					>	>	>	>	>	>	<	<	<	<	<	<	<	<	<	>
>=					>	>	>	>	>	>	<	<	<	<	<	<	<	<	<	>
<					>	>	>	>	>	>	<	<	<	<	<	<	<	<	<	>
<=					>	>	>	>	>	>	<	<	<	<	<	<	<	<	<	>
==					>	>	>	>	>	>	<	<	<	<	<	<	<	<	<	>
!=					>	>	>	>	>	>	<	<	<	<	<	<	<	<	<	>
+					>	>	>	>	>	>	>	>	<	<	<	<	<	<	<	>
-					>	>	>	>	>	>	>	>	<	<	<	<	<	<	<	>
*					>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	>
/					>	>	>	>	>	>	>	>	>	>	<	<	<	<	<	>
float					>	>	>	>	>	>	>	>	>	>					>	>
int					>	>	>	>	>	>	>	>	>	>					>	>
id					>	>	>	>	>	>	>	>	>	>					>	>
(<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	
)					>	>	>	>	>	>	>	>	>	>					>	>
\$					<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	accept

5.4 LL-tabulka

[illegible]

Pozn. O - Operatory

6 Reference

1. Meduna, A., Lukáš, R.: *IFJ, přednášky [online]*. Brno, Dostupné z:
<http://www.fit.vutbr.cz/study/courses/IFJ/public/materials/index.php>
2. Wikipedia: *Hašovací tabulka*. Dostupné z:
https://en.wikipedia.org/wiki/Hash_table
3. Burgetová, I., Hranický, R., Honzík, J.M.: *IAL, přednášky [online]*. Brno.